

# Naval Research Laboratory

Washington, DC 20375-5320



**AD-A256 071**



NRL/MR/5520-92-7136

## System Design and Development of a Low Data Rate Voice (1200 bps) Rate Converter

J. P. HAUSER

*Communications Systems Branch  
Information Technology Division*

September 30, 1992

92-26812



44h

DTIC  
ELECTE  
OCT 09 1992  
S D

92 10 8 046

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 30, 1992	3. REPORT TYPE AND DATES COVERED Interim 7/91-6/92		
4. TITLE AND SUBTITLE System Design and Development of a Low Data Rate Voice (1200 bps) Rate Converter			5. FUNDING NUMBERS PU - 0602232N PR - RC32A13	
6. AUTHOR(S) J. P. Hauser				
7. PERFORMING ORGANIZATION NAME(S) and ADDRESS(ES) Naval Research Laboratory Washington, DC 20375-5320			8. PERFORMING ORGANIZATION REPORT NUMBER NRL/MR/55520-92-7136	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Command, Control and Ocean Surveillance San Diego, CA 92152-5122			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  This report presents both a high level and a detailed design for a low data rate voice Rate Converter (RC). On the transmit side, the converter reduces 2400 bps voice generated by an Advanced Narrowband Digital Voice Terminal (ANDVT) to a 1200 bps bit stream. On the receive side it converts the 1200 bps bit stream back to a 2400 bps stream in ANDVT format. Rate reduction is accomplished with little degradation to the inherent voice quality of the ANDVT.  This primary focus is upon the real-time software design which is implemented using VxWorks, a real-time, multi-tasking operating system and development environment. The high level design defines four tasks, each having its own execution thread and its own "pipe" to facilitate inter-task communication. The Supervisor Task performs initialization and manages input of commands and data to the RC. The Compressor Task reduces a 2400 bps bit stream to 1200 bps while the Decompressor Task converts from 1200 bps back to 2400 bps. The Output Task manages the output of data from the RC. Latter sections of this report describe the software in detail.				
14. SUBJECT TERMS Low data rate voice      Data/voice integration Rate conversion			15. NUMBER OF PAGES 41	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

## CONTENTS

<b>1.0 SCOPE</b> .....	<b>1</b>
1.1 Purpose .....	1
1.2 Communication Support System (CSS) Architectural Context .....	1
1.3 Voice Subscriber Terminal (VST) .....	1
1.4 Document Overview .....	3
<b>2.0 REFERENCED DOCUMENTS</b> .....	<b>4</b>
2.1 Government Documents .....	4
2.2 Nongovernment Documents .....	4
<b>3.0 GLOSSARY</b> .....	<b>5</b>
<b>4.0 HIGH LEVEL SYSTEM DESIGN</b> .....	<b>6</b>
4.1 Hardware Specification .....	6
4.2 Software Design .....	6
<b>5.0 DETAILED SOFTWARE DESIGN</b> .....	<b>10</b>
5.1 Code Layout .....	10
5.2 User Defined Data Types .....	11
5.3 Message Formats .....	15
5.4 Task Modules .....	16
5.5 I/O Devices .....	27
<b>6.0 REAL-TIME IMPLEMENTATION</b> .....	<b>29</b>
<b>7.0 Appendices</b> .....	<b>31</b>
7.1 Overview of the ANDVT Rate Conversion Algorithm .....	31
7.2 Example Compression/Decompression of Four Consecutive ANDVT Frames .....	35

DTIC ORIGINATOR'S REPORT

<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
<b>By</b> _____	
<b>Distribution/</b>	
<b>Availability Codes</b>	
<b>Dist</b>	<b>Avail and/or Special</b>
A-1	

# **SYSTEM DESIGN AND DEVELOPMENT OF A LOW DATA RATE VOICE (1200 bps) RATE CONVERTER**

## **1.0 SCOPE**

### **1.1 Purpose**

This document encompasses both the high level and the detailed design of the Low Data Rate Voice Rate Converter (RC). Both the hardware and the software designs are covered, with primary attention given to the software design, since all RC development work involves software, while the hardware is strictly off-the-shelf.

### **1.2 Communication Support System (CSS) Architectural Context**

The Rate Converter should be understood within the context of the Communication Support System (CSS) architecture. The goal of the CSS is to support a wide variety of Navy applications, e.g., voice, tactical data, record message, etc., by granting access to a single system that manages all the communication resources. The CSS System Specification makes the following statement:

“A cornerstone of the CSS concept is that the users are not aware of the media employed to transfer data to or from other users. The users are also not aware of data rate, coding mechanisms, link protocols, or timing relationships. The users regard the CSS as only providing the required communications services in terms of distribution, security, quality, timeliness, and throughput.”

The CSS architecture includes both satellite and terrestrial RF transmission systems. Satellite channels, links, and networks within CSS have the bandwidth required to support both voice and data applications, but can benefit from the reduced throughput requirements afforded by low data rate voice techniques. However, terrestrial digital RF networks that are characterized by low bandwidths and dynamically varying connectivities, demand the use of low data rate voice techniques as a prerequisite for the support of voice applications. The Rate Converter seeks to meet that demand.

Figure 1, which is redrawn from the CSS System Specification, depicts the CSS architectural context. As the figure illustrates, users always access CSS communication facilities via a Subscriber. For voice applications, a Voice Subscriber Terminal is currently under development.

### **1.3 Voice Subscriber Terminal (VST)**

Advanced Communication Systems (ACS), Inc., developed a preliminary version of a VST under SBIR N89-41, while Naval Research Laboratory (NRL) Codes 5521 and 5531 have developed the RC with funding from the Shared Adaptive Internetworking Technology (SAINT) 6.2 program.

Figure 2 shows the CSS compatible VST now under development by ACS. The VST must support several interfaces: 1) an interface to an Advanced Narrowband Digital Voice Terminal (ANDVT) operating in voice only mode that transmits and receives 2400 bps red

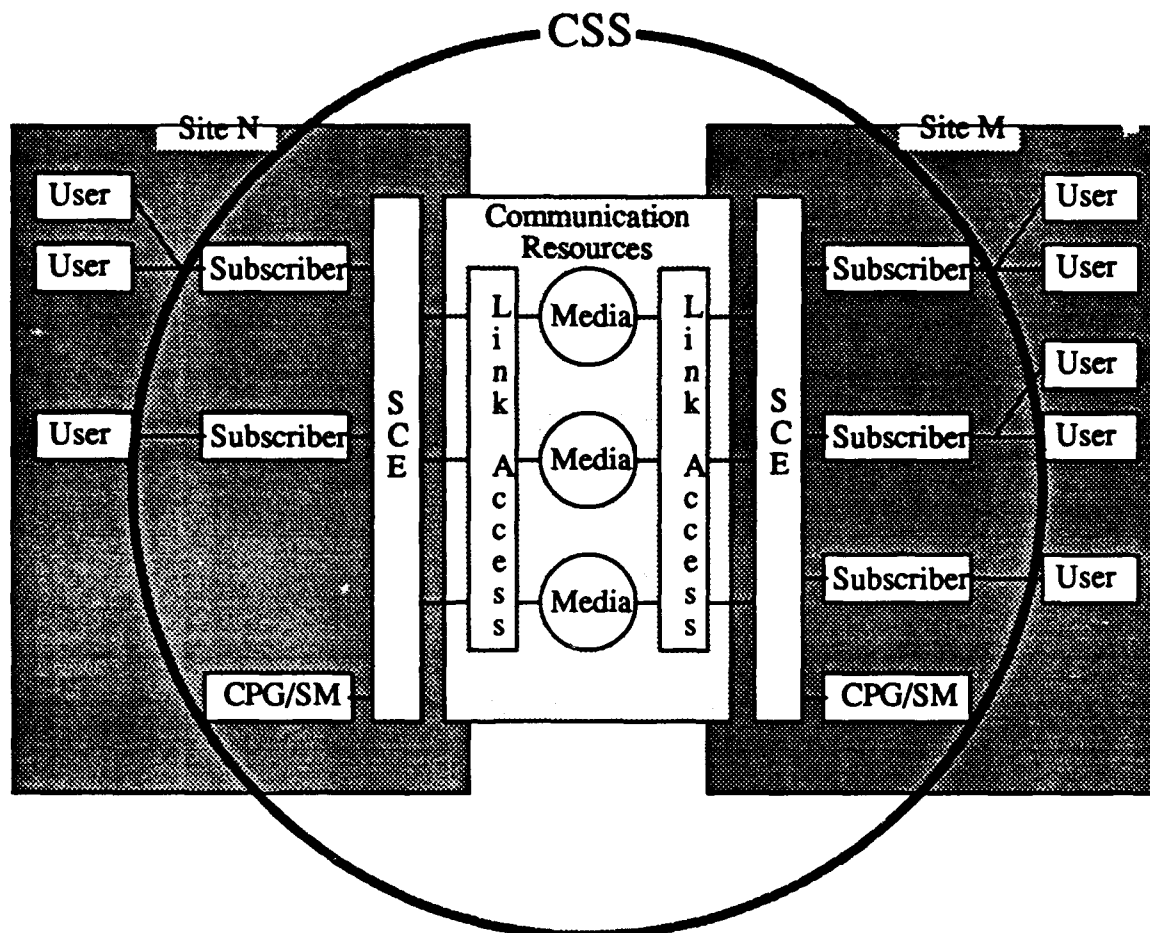


FIGURE 1. CSS Architectural Context.

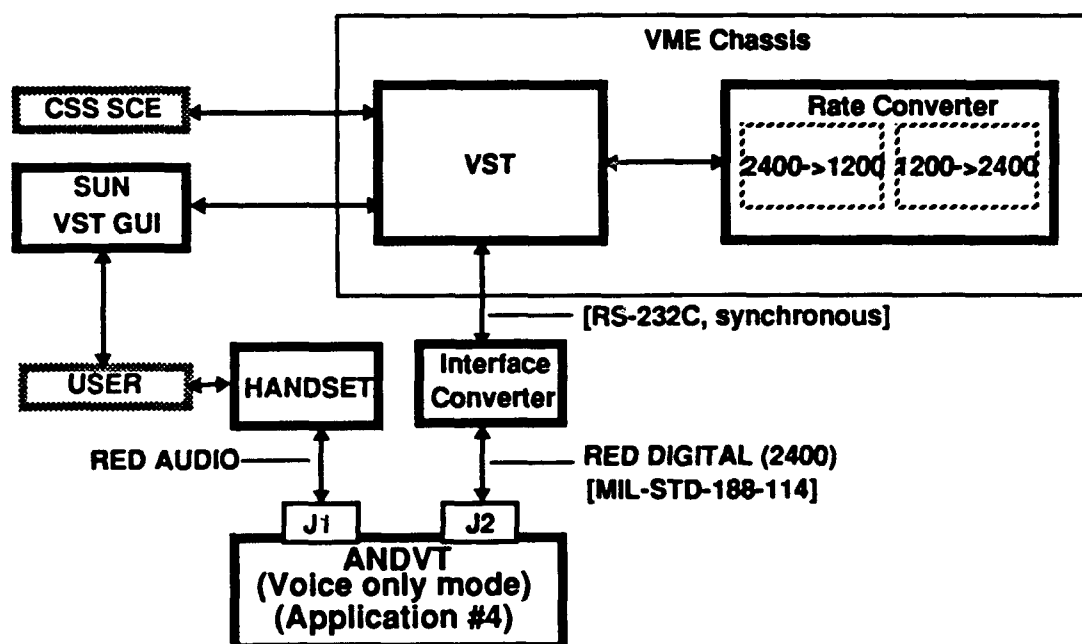


FIGURE 2. CSS compatible Voice Subscriber Terminal (VST) with a Rate Converter (RC) server.

(i.e., unencrypted) voice data via the J2 connector [MIL-C-28883A], 2) a graphical user interface (GUI), 3) an interface to the Standard Communication Environment (SCE), i.e., the remainder of the system, via an Ethernet using a CSS interprocess communication protocol (OS/IPC), and 4) an interface to the RC, which provides 1200 bps voice capability by compressing (for transmission) and decompressing (upon reception) the 2400 bps voice produced by an ANDVT. A broad definition of a CSS Subscriber, as pictured in Figure 1, would include all the elements of Figure 2 as components of the Subscriber, except for the User and the SCE. Thus, the RC, the ANDVT, the Handset, the Interface Converter, the GUI, and the VST are all Subscriber components.

## **1.4 Document Overview**

Having shown where the Rate Converter fits into the context of the CSS, the remainder of this document is devoted to presenting the RC design and implementation. We begin with a high level system design and then proceed to a detailed design of the RC software. We conclude with a discussion of current progress in producing a real-time implementation of the RC.

## 2.0 REFERENCED DOCUMENTS

### 2.1 Government Documents

- G. S. Kang and L. J. Fransen, "ANDVT Rate Conversion Algorithm (From 2400 b/s to 1200 b/s)," NRL Report 9357, September 1991.
- "System Specification for the CSS," NOSC (now NRaD) CSS Program Office, Code 8503, June 1991.
- "System Specification for the CSS Standard Communications Environment (SCE)," NOSC (now NRaD) CSS Program Office, Code 8503, June 1991.
- MIL-C-28883A, "Military Specification for the Advanced Narrowband Digital Voice Terminal (ANDVT) [CV-3591(p)/U (Tactical Terminal)] [J-3953/U (Interface Unit)] [C-11006/U (Modem/Voice Processor Unit)]", Space and Naval Warfare Systems Command, Wash., D.C., 1 June 1987.
- EE160-GP-OMI-010/W151-USC-43, "Technical Manual - Operator and Organizational Maintenance for Advanced Narrowband Digital Voice Terminal - Terminal Sets AN/USC-43(V)1 through AN/USC-43(V)6", ITT Defense Communications Division, 31 July 1987.

### 2.2 Nongovernment Documents

- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Addition, Prentice-Hall, 1988.
- Margaret A. Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- *VxWorks Programmer's Guide*, Wind River Systems, Inc., Alameda, CA
- *X Protocol Reference Manual*, O'Reilly & Associates, Inc., Sebastapol, CA

### **3.0 Glossary**

1. ANDVT - (Advanced Narrow Band Data Voice Terminal)
2. CPG/SM - (Connection Plan Generation / Key Management)
3. CSS - (Communication Support System)
4. GUI - (Graphical User Interface)
5. LPC -10 - (Linear Predictive Code of order 10)
6. MC68020 - Motorola microprocessor that is a first generation full 32 bit machine.
7. MC68030 - Motorola microprocessor that is a second generation full 32 bit machine. It has on-chip caches and multiple internal buses for both data and instructions.
8. MC68040 - Motorola microprocessor that is a third generation full 32 bit machine.
9. MVME135A - VME board-based computer that uses a MC68020.
10. MVME147 - VME board-based computer that uses a MC68030.
11. MVME167 - VME board-based computer that uses a MC68040.
12. OS/IPC - (Operating System / Inter-Process Communication)
13. RC - (Rate Converter)
14. RS-232 - serial interface specification
15. SCC - (Serial Communication Controller)
16. SCE - (Standard Communications Environment)
17. TCP - (Transport Control Protocol)
18. VME - (Virtual Memory)
19. VST - (Voice Subscriber Terminal)
20. VxWorks - real-time, multi-tasking, operating system and accompanying program development environment commercially available from Wind River Systems, Inc.
21. Z8530 - Zilog SCC chip



## 4.0 High Level System Design

### 4.1 Hardware Specification

The Rate Converter design was originally targeted for implementation on a MVME135A card running a VxWorks, real-time, multi-tasking, operating system. This card uses a Motorola MC68020 microprocessor with 4 Mb of onboard RAM. Timing tests have indicated the need to go to a faster board (section 6.0), even after optimizing the C code for rapid execution. It appears that our best hardware option is to implement the RC on a MVME167 board.

Since both the RC and the VST use a common VME bus, the bus will be used to support the RC / VST interface. An Ethernet card (ENP-10L) will be used to support the VST / SCE interface in the target system. In the development environment, the Ethernet is also used as an interface to the Sun Unix host development system to support software downloading and debugging.

### 4.2 Software Design

#### 4.2.1 Task Decomposition

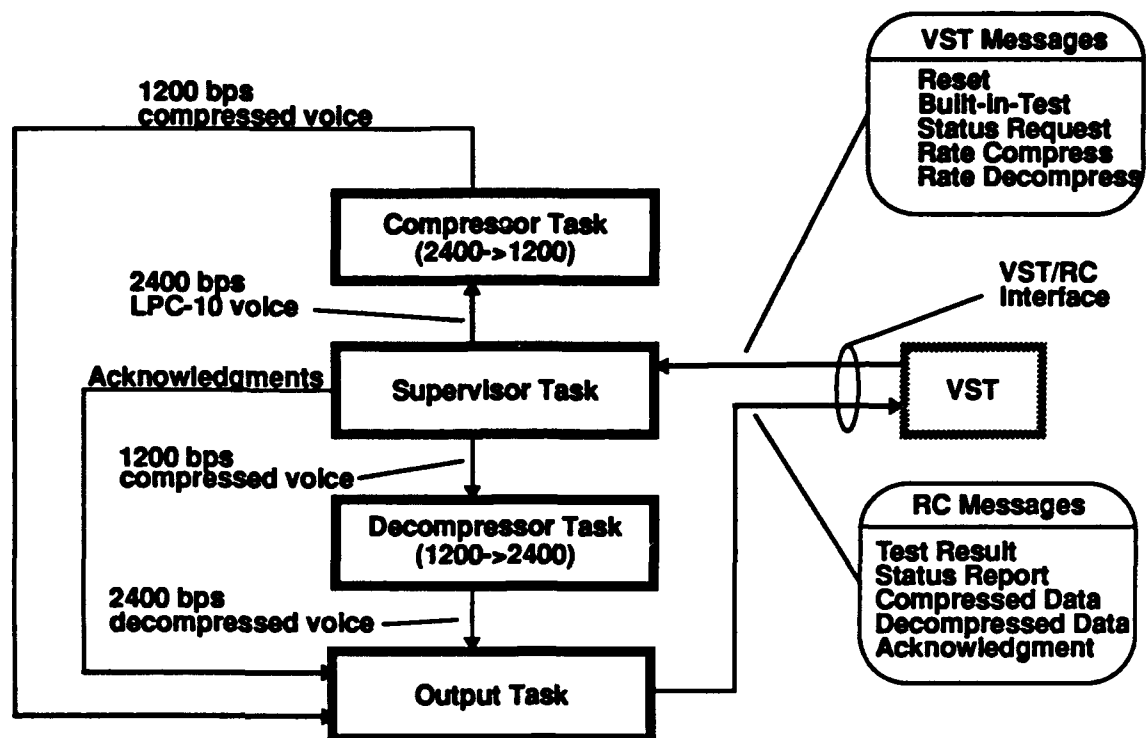
Figure 3 shows a top level RC software design composed of four tasks running under control of the VxWorks multi-tasking operating system. A task in VxWorks is an independent program unit that has its own stack and program counter. Task execution is scheduled by the operating system kernel using preemptive priority-based scheduling. Each task (except the Supervisor Task) is coupled to an input pipe that it reads to get messages sent to it by the other tasks. The Supervisor Task reads its input messages from an input device, which may be either a serial communication port (RS232) located on the microprocessor's front panel or a socket that uses the VME backplane. (For more description of I/O facilities, see section 5.5.)

An important feature of the RC design is the division of input and output handling into separate tasks. This division frees the Supervisor Task from handling both input and output to the VST and, consequently, allowing the call to the read function to temporarily block output. Even with full duplex communication facilities available, this partitioning is necessary to decouple RC input and output.

The functional description of each task follows:

#### 1. Supervisor Task

- Spawn other tasks.
- Create and open pipes (one pipe per task to handle intertask communication).
- Read and decode messages from VST.
- Send ANDVT voice frames (generated locally) to Compressor Task.



**FIGURE 3. Rate Converter (RC) implementation using VxWorks Tasks.**

- Send compressed voice data (compressed by a remote node and received by the local node) to the Decompressor Task.
  - Generate acknowledgments for every message received from the VST and send them to the Output Task.
2. Compressor Task
- Receive messages containing ANDVT data from Supervisor Task.
  - Decode 54 bit ANDVT frames (22.5 ms per frame = 2400 bps) to obtain LPC-10 parameter values.
  - Collect and normalize LPC-10 values from four contiguous ANDVT frames.
  - Test frame arrival time intervals to insure continuity of data.
  - Execute compression algorithm to convert four sets of LPC-10 parameter values to one set of RC parameters.
  - Encode compressed RC parameter values into one 108 bit frame (90 ms per frame = 1200 bps).
  - Assemble 108 bit frame of RC compressed data in a message and send it to the Output Task.
3. Decompressor Task
- Receive RC compressed data (i.e., message containing 108 bit frame) from Supervisor Task.
  - Decode 108 bit frame to obtain RC parameter values for input to decompression algorithm.

- Execute decompression algorithm to obtain four sets of LPC-10 parameter values.
- Encode one 54 bit ANDVT frame per parameter value set.
- Assemble each ANDVT frame in a message and send it to the Output Task.

#### 4. Output Task

- Generate and encode sequence numbers for all RC messages except acknowledgments.
- Send RC messages to VST.

#### 4.2.2 VST/RC Interface

Another important feature of the RC design shown in Figure 3 is the VST/RC interface. The interface is composed of two sets of messages: 1) VST messages sent by the VST to the RC and 2) RC messages sent from the RC to the VST. Using X Window System protocol terminology (appropriate because CSS mandates the X protocol), the VST is the *client* and the RC is the *server* in the client/server paradigm. Likewise, the VST messages are *requests* and the RC messages are *replies*. Table 1 describes these messages.

**Table 1: VST/RC Message Interface**

Type	Sent By	Message Function
Reset	VST	Requests RC to reboot.
Built-in-Test	VST	Requests RC to run a built-in-test procedure.
Status Request	VST	Requests the RC to generate and return a Status Report.
Rate Compress	VST	Requests the RC to compress a 54 bit ANDVT frame. (see section 7.1) Four contiguous ANDVT frames must be received by the RC to generate one 108 bit Compressed Data frame.
Rate Decompress	VST	Requests the RC to decompress a 108 bit Compressed Data frame. The RC regenerates four ANDVT frames.
Test Result	RC	Reply to Built-in-Test message. The RC tests for proper initialization, executes both the Compression and Decompression algorithms with canned input, and compares the results with predetermined values. (Section 7.2 uses the input and output values incorporated in this test.)

**Table 1: VST/RC Message Interface**

Type	Sent By	Message Function
Status Report	RC	<p>Reply to Status Request Message. The following information is included:</p> <ul style="list-style-type: none"><li>• numbers of Rate Compress and Rate Decompress messages read in by Supervisor Task</li><li>• arrival times for the most recent Rate Compress and Rate Decompress messages</li><li>• number of ANDVT frames passed to Compressor Task</li><li>• number of ANDVT frames discarded by Compressor Task</li><li>• difference in arrival times between current and preceding ANDVT frames</li><li>• number of Compressed Data (108 bit) frames passed to Decompressor Task</li></ul>
Compressed Data	RC	<p>Reply to Rate Compress message. Actually, four contiguous Rate Compress messages are required to generate one Compressed Data message reply. Message continuity is defined by a maximum allowable delay from one Rate Compress message to the next (i.e., one ANDVT frame to the next).</p>
Decompressed Data	RC	<p>Reply to Rate Decompress message. Four Decompressed Data messages are generated for each Rate Decompress request.</p>
Acknowledgment	RC	<p>One Acknowledgment is generated by the RC for each VST message received.</p>

## 5.0 Detailed Software Design

### 5.1 Code Layout

The RC source code is written as a set of compilation modules and header files. All the modules, except for one coded in C, are written in C++. Four of the C++ modules define the main routines for VxWorks tasks. The remaining C++ modules define the additional data types, i.e., Bitvectors and Msgs, and a few other miscellaneous functions. The C module, rcConvert.c, defines additional functions that implement the rate conversion algorithms. Compilation files are described in Table 2, header files in Table 3, and other files in Table 4.

**Table 2: RC Compilation Files**

File Name	Type	Description
Misc.C	C++	defines the new and delete operators. These are C++ operators for dynamic memory allocation.
helper.C	C++	defines functions for constructing and destructing C++ globals. These are required since VxWorks does not directly support C++.
rcComp.C	C++	defines the Compressor Task's main program. It also includes functions for decoding ANDVT frames, encoding compressed data frames, and incrementing a clock tick counter.
rcConvert.c	C	defines rate compression and rate decompression algorithms.
rcDecomp.C	C++	defines the Decompressor Task's main program. Also, it defines functions to decode compressed voice data frames and encode ANDVT frames.
rcMsg.C	C++	defines Class Msg (see section 5.2.2).
rcOutput.C	C++	defines the Output Task's main program and a function to compute message sequence numbers.
rcSupv.C	C++	defines the Supervisor Task's main program and functions for the RC's self test and status report.
vx_bitclass.C	C++	defines Class Bitvector (see section 5.2.1) and Class Bitobject.
vx_bitmatrix.C	C++	defines Class Bitmatrix (required for loading Class Bitvector).

**Table 3: Header Files**

File Name	Type	Description
rc.h	C++	provides declarations and compiler directives for the following compilation modules: <ul style="list-style-type: none"> <li>• rcComp.C</li> <li>• rcDecomp.C</li> <li>• rcMsg.C</li> <li>• rcOutput.C</li> <li>• rcSupv.C</li> </ul>
vx_bitclass.h	C++	provides declarations and compiler directives for the following compilation modules: <ul style="list-style-type: none"> <li>• rcComp.C</li> <li>• rcDecomp.C</li> <li>• rcMsg.C</li> <li>• rcOutput.C</li> <li>• rcSupv.C</li> <li>• vx_bitclass.C</li> </ul>
vx_bitmatrix.h	C++	provides declarations and compiler directives for the following compilation modules: <ul style="list-style-type: none"> <li>• vx_bitmatrix.C</li> </ul>

**Table 4: Miscellaneous Files**

File Name	Type	Description
makefile	ASCII	contains commands for the <b>make</b> program.
rc	binary	contains the RC object code and is created by running <b>make</b>
rc.dat	binary	contains initialization data for the reflection coefficient tables used by the rate conversion algorithms.
script	ASCII	contains a script that VxWorks uses after rebooting to load the RC object code and execute a helper function to initialize C++ globals.
vxBoot	ASCII	contains VxWorks boot commands.

## 5.2 User Defined Data Types

We define two additional data types using the object-oriented facilities of C++ - Class Msg and Class Bitvector. A C++ *class* is an extension of a C *structure* that adds executable code and/or member functions to the structure's set of declared variables. These additional data types are quite useful in simplifying the code required to implement the VxWorks tasks outlined in section 4.2.

### 5.2.1 Class Bitvector

Class Bitvector provides bit handling capability. The functions *getbit* and *putbit* use one byte variables (type *char*) having values of zero or one to specify bitvector bit values. Also, *getint* and *putint* provide conversion to positive integer (type *int*) values, while *gettextbits* converts bitvector contents to a string of ascii 1's and 0's.

In addition to the direct bit handling capabilities, Class Bitvector supports the concept of a *field*. Bitvectors can be subdivided into fields by giving the starting bit position and the length of the field within the bitvector. Fields are also bitvectors and, therefore, have the same bit handling capabilities. Fields provide a natural way to specify and manage message formats and their contents.

Bitvectors are used extensively by the Compressor and Decompressor Tasks to decode and encode ANDVT frames. Also, bitvectors are used to implement Class Msg since messages are formatted on the bit level. The interface to Class Bitvector is presented in Table 5.

Table 5: Class Bitvector Interface

Member Function Declaration	Function Description
<code>bitvector ()</code>	constructor used to initialize a bitvector.
<code>bitvector (int sz)</code>	alternate constructor that takes the bitvector's length in bits as input.
<code>~bitvector ()</code>	destructor used to garbage collect a bitvector.
<code>void array_initializer (int sz)</code>	same as alternate constructor - used for array initialization.
<code>void attach (int *buf, int size_in_bits)</code>	buf, of length size_in_bits, becomes the contents of the bitvector.
<code>int start ()</code>	returns the starting position of the bitvector in the buffer that holds it.
<code>int length ()</code>	returns the length of the bitvector in bits.
<code>char getbit (int i)</code>	returns the value of the bit at bit position <i>i</i> in the bitvector (first position = 0).
<code>void putbit (int i, char b)</code>	puts a bit of value <i>b</i> at position <i>i</i> in the bitvector.
<code>char *gettextbits (char *dest, int size_of_dest)</code>	places a character ('1' and '0') representation of the bitvector's contents into <i>dest</i> .

**Table 5: Class Bitvector Interface**

Member Function Declaration	Function Description
<code>bitvector *field (int i1, int i2)</code>	returns a pointer to a new bitvector that accesses a portion of this bitvector's contents (i.e., a field) starting at bit position <u>i1</u> and extending for <u>i2</u> bits in length.
<code>int getint (int num_bits_in_field = -1, int pos_of_first_bit=0)</code>	converts the binary number starting at <u>pos_of_first_bit</u> and extending for <u>num_bits_in_field</u> to a positive integer and returns the value (default: entire contents of bitvector if no input parameters given).
<code>void putint (int i, int num_bits_in_field = -1, int pos_of_first_bit=0)</code>	converts a positive integer to binary and inserts it in bitvector beginning at <u>pos_of_first_bit</u> and extending for <u>num_bits_in_field</u> .
<code>void setallbits ()</code>	sets all bits to '1'.
<code>void clearallbits ()</code>	clears all bits to '0'.
<code>void setbit (int i)</code>	sets bit at position <u>i</u> to '1'.
<code>void clearbit (int i)</code>	clears bit at position <u>i</u> to '0'.
<code>void assign (bitvector *b_v, int num_bits_to_copy=-1, int pos_of_first_bit_to_copy = 0, int pos_of_dest_bit = 0)</code>	copies contents of <u>b_v</u> , starting at <u>pos_of_first_bit_to_copy</u> and extending for <u>num_bits_to_copy</u> , into this bitvector, starting at <u>pos_of_dest_bit</u> .
<code>bitvector *and_bits (bitvector *bv)</code>	replaces the contents of this bitvector with the bitwise logical <i>and</i> of this bitvector and <u>bv</u> (bitvector lengths must be equal).
<code>bitvector *or_bits (bitvector *bv)</code>	replaces the contents of this bitvector with the bitwise logical <i>or</i> of this bitvector and <u>bv</u> (bitvector lengths must be equal).
<code>bitvector *ones_complement ()</code>	replaces the contents of this bitvector with the ones complement of this bitvector.
<code>char is_zero ()</code>	returns TRUE if all bits are '0'.
<code>char same_as (bitvector *bv)</code>	returns TRUE if this bitvector and <u>bv</u> have the same contents.



**Table 5: Class Bitvector Interface**

Member Function Declaration	Function Description
<code>int state()</code>	returns one of three state values <ul style="list-style-type: none"> <li>• <code>BV_IS_MAIN</code> (= 0)</li> <li>• <code>BV_IS_NOT_MAIN</code> (= 1)</li> <li>• <code>BV_IS_UNATTACHED</code> (= 2)</li> </ul>

**5.2.2 Class Msg**

A high level description of the message set that implements the RC / VST interface appears in section 4.2.2. Class Msg hides the low level details of handling messages behind the interface given in Table 6. The same messages used externally to support the RC / VST interface are used internally for intertask communication via pipes.

**Table 6: Class Msg Interface**

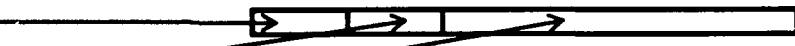


Member Function Declaration	Function Description
<code>Msg ()</code>	constructor used for initializing a msg.
<code>~Msg ()</code>	destructor used to garbage collect a msg.
<code>void setContents (MsgType tp,           unsigned char sn,           int ln,           bitvector *bv=NULL)</code>	sets the contents of msg by providing values for msg type, sequence number, msg length (bytes), and data (default: if <i>bv</i> is not given, msg has no data, but the msg type, the sequence number, and the msg length in <i>bytes</i> must be given).
<code>void setLength (int ln)</code>	sets the msg length to <i>ln</i> bytes = 1 byte msg type + 1 byte sequence number + <i>n</i> bytes data).
<code>void setSeqnum (unsigned char sn)</code>	sets the msg sequence number to <i>sn</i> (0 - 255).
<code>bitvector *getContents ()</code>	returns a pointer to the contents of msg (type + sequence number + data).
<code>MsgType getType ()</code>	returns the type of msg.
<code>unsigned char getSeqnum ()</code>	returns the sequence number of msg.
<code>int getLength ()</code>	returns the length in bytes of msg.
<code>bitvector *getData ()</code>	returns a pointer to the data of msg.
<code>void display (int fd)</code>	displays msg on device designated by file descriptor <i>fd</i> .

Table 6: Class Msg Interface

Member Function Declaration	Function Description
int mread (int fd)	reads contents of msg from stream oriented device designated by file descriptor <u>fd</u> . (see section 5.5)
int pread (int fd)	reads contents of msg from message oriented device (e.g., VxWorks pipe) designated by file descriptor <u>fd</u> .
int mwrite (int fd)	writes contents of msg to device designated by file descriptor <u>fd</u> .

### 5.3 Message Formats

All message formats used by the RC have the following three fields:

- message type (1 byte), 
- sequence number (1 byte), 
- data (0 - 35 bytes). 

The message type field is one byte in length. Message type values are given by the following enumeration:

```
enum MsgType {
    RESETmsg,    /* VST->RC: RC reboots */
    TESTmsg,     /* VST->RC: RC runs test */
    STATUSmsg,   /* VST->RC: RC returns status */
    COMPmsg,     /* VST->RC: RC compresses 2400 data */
    DECOMPmsg,   /* VST->RC: RC decompresses 1200 data */
    RESULTmsg,   /* RC->VST: test result */
    REPORTmsg,   /* RC->VST: status report */
    COMDDmsg,    /* RC->VST: compressed data */
    DECOMDDmsg,  /* RC->VST: decompressed data */
    ACKmsg,      /* RC->VST: acknowledgment */
};
```

The sequence number field is an `unsigned char` that has a decimal value in the range 0 - 255. The length and contents of the data field depends on the type of message, as shown in Table 7.

Table 7: Data Field Formats

MsgType	Msg Length (bytes)	Data Field Range (bits)	Data Field Format
RESETmsg	2	0	none

**Table 7: Data Field Formats**

<b>MsgType</b>	<b>Msg Length (bytes)</b>	<b>Data Field Range (bits)</b>	<b>Data Field Format</b>
<b>TESTmsg</b>	2	0	none
<b>STATUSmsg</b>	2	0	none
<b>COMPmsg</b>	9	0 - 53	54 bit ANDVT frame
<b>DECOMPmsg</b>	16	0 - 107	108 bit compressed voice data frame
<b>RESULTmsg</b>	3	0 - 2	<ul style="list-style-type: none"> <li>• (0) error in table initialization</li> <li>• (1) error in rc24to12 execution</li> <li>• (2) error in rc12to24 execution</li> </ul>
<b>REPORTmsg</b>	35	0 - 263	<ul style="list-style-type: none"> <li>• (0 - 7) TRUE if pipe &amp; task initialization OK</li> <li>• (8 - 39) number of <b>COMPmsg</b>s read in by Supervisor Task</li> <li>• (40 - 71) number of <b>DECOMPmsg</b>s read in by Supervisor Task</li> <li>• (72 - 103) clock tick count when most recent <b>COMPmsg</b> read in</li> <li>• (104 - 135) clock tick count when most recent <b>DECOMPmsg</b> read in</li> <li>• (136 - 167) number of <b>COMPmsg</b>s read in by Compressor Task</li> <li>• (168 - 199) number of <b>COMPmsg</b>s discarded by Compressor Task</li> <li>• (200 - 231) number of clock ticks for most recent <b>COMPmsg</b> arrival interval</li> <li>• (232 - 263) number of <b>DECOMPmsg</b>s read in by Decompressor Task</li> </ul>
<b>COMDDmsg</b>	16	0 - 107	108 bit compressed voice data frame
<b>DECOMDDmsg</b>	9	0 - 53	54 bit ANDVT frame
<b>ACKmsg</b>	2	0	none

## 5.4 Task Modules

Subsection 4.2.1 provides functional descriptions for each of the task modules. In the following subsections we outline the code required to implement each VxWorks task module and the rate conversion algorithm module. Each task module contains the following elements:

- **Include files** - header files that contain declarations and `#define` statements. Both `#include` and `#define` are C preprocessor statements that can be viewed as a first step in compilation. `#include` copies the contents of the included file into the module's source code. Similarly, `#define` replaces tokens in the source code with replacement text.
- **External declarations** - `extern` is a C keyword that introduces a variable or function that is defined in an external module (i.e., file) into the scope of the module using the `extern` declaration. Because the RC source code is partitioned into several modules, the use of the `extern` declaration is required to give a module access to certain variables and functions defined in other modules. Placing the external declarations directly in the source code rather than in a header file helps to show the external variables and functions that a module depends on.
- **Global variables** - variables that have global scope within the module, i.e., are defined outside of any function and are accessible by every function in the module.
- **Functions** - most are called by the main program of the module in which their definitions appear, but a few are designed to be called from other modules. For example, `rcinit()` is defined in `rcConvert.c` because it initializes arrays that are likewise defined in `rcConvert.c`. However, `rcinit()` is called by the main program of `rcSupv.C`.
- **Main program** - a module's main routine is defined just like any other function. The difference is that it is executed by a call to the VxWorks system function `spawn()`. A call to `spawn()` gives the function its own program counter and stack, thus making it a vx-Works *task*. In the following subsections, we describe main programs in terms of pseudo code.

### 5.4.1 Supervisor Task (rcSupv.C)

Include files:

pathname for VxWorks  
on host system

These files are included if sockets are to be used for RC/  
VST communication. Otherwise, RS232 ports are used.

```
extern "C" {  
#include    "/home/nautilus2/hauser/vw502/h/vxWorks.h"  
#include    "/home/nautilus2/hauser/vw502/h/sysLib.h"  
#ifndef RS232  
#define __STDC__  
#include    "/home/nautilus2/hauser/vw502/h/types.h"  
#include    "/home/nautilus2/hauser/vw502/h/in.h"  
#include    "/home/nautilus2/hauser/vw502/h/sockLib.h"  
#include    "/home/nautilus2/hauser/vw502/h/socket.h"  
#endif  
}  
#include    "vx_bitclass.h"  
#include    "rc.h"
```

extern "C" { } is re-  
quired to prevent C++  
from renaming these in-  
clude files to C++  
names different from  
the C names that #in-  
clude will look for.

These files will be renamed to C++ names,  
which is fine since they are C++ files.

## External declarations:

```

extern "C" void rcCompMain __Fv 0;
extern "C" void rcDecompMain __Fv 0;
extern "C" void rcOutputMain __Fv 0;
extern "C" void rcinit 0;
extern "C" char* rcTest();
extern "C" double ceil (double);
extern "C" int pipeDevCreate (char*, int, int);
extern "C" int open (char*, int, ...);
extern "C" int taskSpawn (char*, ...);
extern "C" void reboot (int);
extern "C" int printErr (char *fmt, ...);
extern "C" int printf (cmachar *fmt, ...);
#ifdef RS232
extern "C" void bzero (char *b, int length);
extern "C" int close (int);
#endif
extern int clkCnt;

```

C++ names for Compressor, Decompressor, and Output Task main programs.

defined in rcConvert.c

defined by VxWorks

defined in rcComp.C

needed for sockets

## Globals:

```

/* GLOBALS */
int compPipeFd; /* file descriptor for comp pipe */
int decompPipeFd; /* file descriptor for decomp pipe */
int outputPipeFd; /* file descriptor for output pipe */
int vstFd; /* file descriptor for RS-232 port or socket */
/*****
 *
 * global status variables
 */
struct rcStatus {
    char supvInitOK; /* TRUE if pipes & tasks init OK */
    int supvNumCOMPmsgsIn; /* num COMP msgs read in by supvMain */
    int supvNumDECOMPmsgsIn; /* num DECOMP msgs read in by supvMain */
    int supvCOMPmsgTic; /* tic cnt (20ms/tic) of most recent COMP msg */
    int supvDECOMPmsgTic; /* tic cnt (20ms/tic) of most recent DECOMP msg */
    int compNumFramesRd; /* number of frames read in by compMain */
    int compDiscardCnt; /* number of frames discarded by compMain */
    int compFrameDelay; /* number of tics from previous frame to current frame */
    int decompNumFramesRd; /* number of frames read in by decompMain */
} stat = {FALSE, 0, 0, 0, 0, 0, 0, 0, 0};

```

File descriptors for all task input pipes and the communication socket (or port) are defined in rcSupv.C.

rcStatus is a structure that contains status information for each of the tasks and is defined in rcSupv.C.

### Functions:

- **void doTest (Msg& result)** - called by rcSupvMain() in response to a TESTmsg request from the VST. doTest() in turn calls rcTest() (defined in rcConvert.c) to run the self test and places the results of the self test into the Msg supplied as a parameter of doTest().
- **void getStatus (Msg& report)** - called by rcSupvMain() in response to a STATUSmsg. getStatus() encodes the contents of an rcStatus structure into the REPORTmsg supplied as a parameter to getStatus().

### Pseudo code for rcSupvMain():

```
create and open input pipes for
    > Compressor Task
    > Decompressor Task
    > Output Task

if (RS232 defined)
    open a raw RS232 port at 9600 baud for communication with VST
else
    open a stream socket for communication with VST
    listen for a connection
    accept a connection
    spawn tasks
        > Compressor Task
        > Decompressor Task
        > Output Task

call initialization rcinit() - reads in rc.dat file

check for successful creation of pipes and tasks and store result in status variable

main loop
    read a Msg from input device (RS232 port or socket)
    if (read successful)
        write ACKmsg to Output Task's input pipe
        if (RESETmsg)
            reboot the RC
        else if (TESTmsg)
            do self test
            write RESULTmsg to Output Task's input pipe
        else if (STATUSmsg)
            get a status report
            write REPORTmsg to Output Tasks's input pipe
        else if (COMPmsg)
            update COMPmsg count
            update COMPmsg time tick
            write COMPmsg to Compressor Task's input pipe
```

LISTEN →

Spawn is a VxWorks command that creates and activates a task. An active task can be scheduled to execute.

```

else (DECOMPmsg)
    update DECOMPmsg count
    update DECOMPmsg time tick
    write DECOMPmsg to Decompressor Task's input pipe
else (read not successful)
    print error
    if (RS232 not defined)
        close socket
        go to LISTEN →
end main loop

```

#### 5.4.2 Compressor Task (rcComp.C)

Include files:

```

extern "C" {
#include    "/home/nautilus2/hauser/vw502/h/vxWorks.h"
}
#include    "vx_bitclass.h"
#include    "rc.h"

```

External declarations:

```

extern int compPipeFd; /* file descriptor for comp pipe */
extern int outputPipeFd; /* file descriptor for output pipe */
extern "C" STATUS sysAuxClkConnect(FUNCPTR,int);
extern "C" void sysAuxClkRateSet(int);
extern "C" void sysAuxClkEnable();
extern "C" void rc24to12(short*,short*); /* subroutine for 2400 to 1200 */
extern "C" int printf(char *fmt, ...); VxWorks function
extern struct rcStatus {...} stat;

```

rcCompMain() reads from compPipeFd and writes to outputPipeFd.

VxWorks functions used to set up a clock.

Globals:

```

int clkCnt = 0; /* count sys aux clk ticks */
int bitkeyVcd[12][7] = {...};
int bitkeyUnvcd[12][9] = {...};

```

Used to count system auxiliary clock ticks.

The values stored in these arrays give the bit positions of LPC-10 parameters in an ANDVT frame.



Globals continued:

Bitvectors used to assemble the bits extracted from ANDVT frames.

```
bitvector p(7); /* scratch bitvector for decoding pitch */
bitvector vcd5(5); /* scratch bitvector for decoding voiced params */
bitvector vcd4(4); /* scratch bitvector for decoding voiced params */
bitvector vcd3(3); /* scratch bitvector for decoding voiced params */
bitvector vcd2(2); /* scratch bitvector for decoding voiced params */
bitvector unvcd5(5); /* scratch bitvector for decoding unvoiced params */
bitvector f12(108); /* frame of 1200 data (108 bits) */
bitvector *drpdFrm = f12.field(0, 1); /* dropped frame (2 bits) */
bitvector *rc1to4Frm1 = f12.field(2, 14); /* rc1 to rc4 frm1 (13 bits) */
bitvector *rc5to10Frm1 = f12.field(15, 27); /* rc5 to rc10 frm1 (13 bits) */
bitvector *rc1to4Frm2 = f12.field(28, 40); /* rc1 to rc4 frm2 (13 bits) */
bitvector *rc5to10Frm2 = f12.field(40, 53); /* rc5 to rc10 frm2 (13 bits) */
bitvector *rc1to4Frm3 = f12.field(54, 66); /* rc1 to rc4 frm3 (13 bits) */
bitvector *rc5to10Frm3 = f12.field(67, 79); /* rc5 to rc10 frm3 (13 bits) */
bitvector *pitch = f12.field(80, 86); /* pitch (7 bits) */
bitvector *ampFrm1 = f12.field(87, 91); /* amplitude frm1 (5 bits) */
bitvector *ampFrm2 = f12.field(92, 96); /* amplitude frm2 (5 bits) */
bitvector *ampFrm3 = f12.field(97, 101); /* amplitude frm3 (5 bits) */
bitvector *vcDrpdFrm = f12.field(102, 102); /* voicing drpd frm (1 bit) */
bitvector *vcFrm1 = f12.field(103, 103); /* voicing frm1 (1 bit) */
bitvector *vcFrm2 = f12.field(104, 104); /* voicing frm2 (1 bit) */
bitvector *vcFrm3 = f12.field(105, 105); /* voicing frm3 (1 bit) */
bitvector *sync = f12.field(106, 107); /* synchronization (2 bits) */
int synchbits = 0; /* set synchbits alternately to 0 (==00) or 3 (==11) */
```

The value of synchbits is used to set the last field of f12 alternately to binary 00 or 11.

f12 is a bitvector containing 108 bits. It is partitioned into fields by the bitvector pointers defined following it. Parameters generated by rate compression are encoded into these fields, and, thus, into f12.

#### Functions:

- void decodeF24 (bitvector \*f24, short \*vp) - decodes the LPC-10 parameters from an ANDVT frame passed in via f24 and places the results at vp.
- void encodeF12 (short \*output) - encodes 1200 bps compressed voice parameters passed in via output into the globally defined bitvector, f12.
- void clkTick () - increments a counter every time it is called by the VxWorks auxiliary system clock.

Pseudo code for rcCompMain():

```
connect the VxWorks auxiliary system clock (AuxClk) to clkTick()
set the AuxClk rate to 50 ticks/s, i.e., 20 ms per tick
enable the clock, i.e., start AuxClk
define automatic variables used in this main program
    set frameCnt to 0 - used to count four 'contiguous' ANDVT frames for compression
main loop
    save current number of clock ticks in tic
    read a Msg (i.e., COMPmsg) from this task's input pipe
    update status variables
        > compNumFramesRd - running sum of number of ANDVT frames read in
        > compFrameDelay - time (i.e., number of 20 ms ticks) that the main loop was
            blocked waiting to read a Msg from the input pipe
    if (compFrameDelay is greater than 10 ticks, i.e., 200 ms)
        update compDiscardCnt (running sum of the number of ANDVT frames discarded
            because of a lack of continuity, i.e., > 200 ms gap between frames)
        set frameCnt back to 0
    call decodeF24() to decode the ANDVT frame just read in and put the decoded
    LPC-10 parameter values into a vector of the array called input
    if (frameCnt equals 3, i.e., four contiguous ANDVT frames have been decoded)
        call rc24to12() to compress four sets of LPC-10 parameters into one set of rate
            compressed parameters
        call encode12() to encode the rate compressed parameters into one 108 bit
            frame
        place the contents of the 108 bit frame into a COMDDmsg
        write the COMDDmsg to the Output Task's input pipe
    increment frameCnt
    if (frameCnt is greater than 3)
        set frameCnt to 0
end main loop
```

#### 5.4.3 Decompressor Task (rcDecomp.C)

Include files:

```
extern "C" {
#include    "/home/nautilus2/hauser/vw502/h/vxWorks.h"
}
#include    "vx_bitclass.h"
#include    "rc.h"
```

## External declarations:

rcDecompMain() reads from decompPipeFd and writes to outputPipeFd.

```
extern int decompPipeFd; /* file descriptor for decomp pipe */
extern int outputPipeFd; /* file descriptor for output pipe */
extern int bitkeyVcd[12][7]; /* decoding key for voiced */
extern int bitkeyUnvcd[12][9];
extern "C" const short i84enc[16];
extern "C" void rc12to24(short*, short*); /* subroutine for 1200 to 2400 */
extern "C" int printf (char *fmt, ...);
extern struct rcStatus {...} stat;
```

defined in rcComp.C -  
used here to encode  
ANDVT frames

## Globals:

```
bitvector ep(7); /* scratch bitvector for encoding pitch */
bitvector evcd5(5); /* scratch bitvector for encoding voiced params */
bitvector evcd4(4); /* scratch bitvector for encoding voiced params */
bitvector evcd3(3); /* scratch bitvector for encoding voiced params */
bitvector evcd2(2); /* scratch bitvector for encoding voiced params */
bitvector eunvcd5(5); /* scratch bitvector for encoding unvoiced params */
bitvector eham4(4); /* scratch bitvector for encoding hamming codes */
```

These bitvec-  
tors are used by  
encodeF24() to  
encode AND-  
VT frames.

## Functions:

- void decodeF12 (bitvector \*f12, short \*op) - decodes one 108 bit frame of compressed voice data and places the compressed voice parameters in a vector pointed to by op.
- void encodeF24 (int j, short \*out, bitvector \*an) - takes the output (pointed to by out) of rc12to24() and encodes one ANDVT frame (pointed to by an).

## Pseudo code for rcDecompMain():

```
define variables global to Decompressor Task's main program
main loop
  read a Msg from this task's input pipe
  increment decompNumFramesRd, the number of compressed voice data frames
  read in for decompression
  call decodeF12() to decode a frame of rate converted data
  call rc12to24() to decompress the frame of rate converted data
  for loop (4 iterations)
    call encodeF24 to encode one ANDVT frame
    put contents of ANDVT frame into DECOMDDmsg
    write DECOMDDmsg to Output Task's input pipe
  end for loop
end main loop
```

#### 5.4.4 Output Task (rcOutput.C)

Include files:

```
extern "C" {  
#include    "/home/nautilus2/hauser/vw502/h/vxWorks.h"  
}  
#include    "vx_bitclass.h"  
#include    "rc.h"
```

External declarations:

```
extern int outputPipeFd;    /* file descriptor for output pipe */  
extern int vstFd;          /* file descriptor for RS-232 port or socket */  
extern "C" int open (char*, int, ...);  
extern "C" int ioctl (int fd, int function, ...);  
extern "C" int printf (char *fmt, ...);
```

Functions:

- unsigned char getSeqNum () - returns a new sequence number value.

Globals:

```
/* GLOBALS */  
unsigned char seqNum = 0;    /* counter used to generate msg sequence #s */
```

Pseudo code for rcOutputMain():

define variables

outFd is initialized to the external, vstFd (This works fine if vstFd is a socket.)

if (RS232 defined, i.e., vstFd is a RS232 port in read only mode)

reinitialize outFd as a RS232 port in write only mode

main loop

read a Msg from the Output Task's input pipe

if (Msg is not an ACKmsg)

set the Msg's sequence number field with a value obtained by calling getSeqNum()

write the Msg to outFd

end main loop

### 5.4.5 Rate Conversion Algorithms (rcConvert.c)

This file contains the functions that implement the rate conversion algorithms. These algorithms are fully described in section 2.1 of [Kang]. rcConvert.c has no main program. Rather, its functions are called by the tasks defined in other code modules.

The rate conversion algorithms were originally written in Fortran. For the sake of compactness and efficiency, they have been rewritten in C. However, the original Fortran code is still retained as comments within the C code.

Include files:

```
#include "/home/nautilus2/hauser/vw502/h/vxWorks.h"
#include "/home/nautilus2/hauser/vw502/h/ioLib.h"
```

Globals:

```
short it1[8192][4];
short it2[8192][4];
short it3[8192][6];
short it1x[2][31];
short it2x[2][31];
short it3x[2][16];

it1, it2, and it3 are arrays of reflection coefficients. They are
the only valid coefficient values for 1200 bps rate converted
voice.
• it1 - unvoiced, 8192 sets of  $C_1 - C_4$ 
• it2 - voiced, 8192 sets of  $C_1 - C_4$ 
• it3 - voiced, 8192 sets of  $C_5 - C_{10}$ 

it1x, it2x, and it3x hold indexes into the coefficient arrays.

const short i84enc[16] = {0,7,11,12,13,10,6,1,14,9,5,2,3,4,8,15};  /
* make global to accommodate rcDecomp.C */
```

Functions:

- void rcinit () - initialization function that reads a binary file (rc.dat) to initialize the globals given above.
- void rc24to12 (input, output) - function that implements the rate compression algorithm. Four sets of LPC-10 parameter values are passed in via input and one set of 1200 bps rate converted voice parameter values are passed out via output.
- void rc12to24 (in, out) - function that implements the rate decompression algorithm. One set of 1200 bps rate converted voice parameter values are passed in via in and four sets of LPC-10 parameter values are passed out via out.
- int pchdec (ipitchx) - decodes an error coded value for pitch, ipitchx, and returns an uncoded value.
- int fixrc (ircx, icorr) - use hamming code to fix a reflection coefficient value that contains bit errors.

- int fixamp (iamp, icorr) - use hamming code to fix an amplitude value that contains bit errors.
- char\* rcTest () - function designed to test the correct operation of the rate conversion algorithms. The return value is a pointer to a character array of length 3 with the following meaning:
  - char [0] - TRUE if global arrays (it1 - it3x) have been properly initialized.
  - char [1] - TRUE if a call to rc24to12() with test input data produces the correct output.
  - char [2] - TRUE if a call to rc12to24() with test input data produces the correct output.

## 5.5 I/O Devices

The read and write functions defined in Class Msg (mread(), pread(), and mwrite()) interface to VxWorks I/O devices. VxWorks and Unix devices have nearly identical interfaces. However, each of the devices discussed in the following subsections have some unique characteristics.

### 5.5.1 Pipes

VxWorks pipes are message oriented devices rather than byte stream oriented devices. This means that a Vxworks pipe must be written to and read from a message at a time. Actually, this is an advantage when implementing a message based design, as we have done in the RC. When a VxWorks pipe is read, we are free to specify the maximum possible message length for the number of bytes in the read request. Only the number of bytes actually in the message will be read. This feature makes it easy to read and process messages from pipe devices. Thus, the code for the pread() member function of Class Msg is less complex than the code for byte stream oriented devices.

### 5.5.2 Serial Port

If the symbolic constant RS232 is defined using a **#define** C preprocessor command, the RC source modules will be compiled for RS-232 communication with a VST. A port is a byte stream oriented device. When reading from such a device, the problem of knowing how many bytes to read must be handled. The approach taken by the Class Msg mread() member function is to read the first byte in order to determine the message type and, by inference, the message length. Then the remainder of the message is read.

### 5.5.3 Stream Socket

If the symbolic constant RS232 is not defined, the RC source modules will be compiled using a stream socket for communication with a VST. In this instance, the VST will also have to create a stream socket and initiate a connection with the RC. The RC in its roll as a server, will listen for and accept a connection from a VST in its roll as a client. The advantage of using sockets as opposed to RS-232 devices is that the protocol (TCP) that supports stream sockets can be run over the VME backplane, or even over a network, i.e.,

Ethernet, if that is deemed appropriate. Assuming the design presented in figure 2 is implemented, the VST and RC will communicate over the VME chassis backplane.

There are a couple of differences between using sockets and using ports. The first is that two ports are opened to the RC's RS-232 device, one for reading and one for writing. On the other hand, the same full-duplex RC socket is used for both reading and writing. The second difference is that a reset request (RESETmsg) from the VST will break the connection between the RC and VST sockets. This connection must be reestablished after the RC has rebooted. On the other hand, an RS-232 connection is always there, unless the RS-232 cable is unplugged or the RS-232 ports are otherwise physically disconnected. Therefore, a reset request behaves in a simple and straightforward manner if RS-232 ports are being used.

## 6.0 Real-Time Implementation

C is used for the Rate Compression and Rate Decompression algorithms. These algorithms were originally coded in Fortran, but for the sake of run-time efficiency they were recoded in C. The rate compression algorithm, implemented in the function `rc24to12()`, is by far the most CPU intensive component of the RC. In particular, the binary tree search through the set of reflection coefficient vectors to find the one that most nearly matches the ANDVT generated vector is the computational loop that uses the bulk of CPU time.

To ascertain the ability of the RC to run in real time, we have concentrated our efforts on timing the execution of the `rc24to12()` function. Also, we have made efforts to further improve the efficiency of the C code, particularly in the aforementioned search loop. The results of our timing tests are given in Table 8.

Table 8: Execution Times for Calls to `rc24to12()`

VME Board	Processor	C Compiler	Execution Time (ms)	Comments
MVME147SA-1	MC68030	AT&T	247	analyzer measurement
MVME167C	MC68040	AT&T	117	analyzer measurement
MVME135A	MC68020	Gnu	222	timexN() measurement
MVME135A	MC68020	Gnu	147	timexN() measurement (revised C code)

The RC software, i.e., the unimproved version of `rc24to12()` function, was benchmarked at Motorola using a MVME147SA-1 and a MVME167C board in conjunction with a board analyzer that measured the entry and exit times from the `rc24to12()` function. Benchmarks on the MVME135A board, i.e., our development system, were performed using the VxWorks `timexN()` system function, which called the `rc24to12()` function multiple times to get an accurate average value for execution time.

The execution times shown in Table 8 are surprising when one considers that the MIPS rate approximately doubles going from a MVME135 to a MVME147 board and, again, doubles going from a MVME147 to a MVME167 board. The comparatively high performance of the MVME135 board is due to the use of the Gnu C compiler instead of the AT&T C compiler. Also, our efforts in speeding up the C code paid dividends, but was not good enough to reduce execution time to less than 90 ms, which must be done for the RC to meet real-time requirements. Since an ANDVT produces one 54 bit frame of data every 22.5 ms (= 2400 bps), the RC must compress four frames every 90 ms to keep up. Also, the RC needs a little time for doing other chores, so a more realistic timing requirement would be < 80 ms. It appears that a MVME147 board with Gnu compiled code would be adequate to run the RC in real time, assuming the Gnu compiler improves the MVME147's performance by a 2:1 factor and our code improvements work well on the MVME147. A safer bet would be to use a MVME167 board. The additional \$500 in cost



(for a MVME167 compared to a MVME147) could mean the difference between marginal and rock solid performance.

## 7.0 Appendices

### 7.1 Overview of the ANDVT Rate Conversion Algorithm

The following overview is extracted directly from material presented in [Kang]. For a comprehensive discussion of the ANDVT Rate Conversion Algorithm, we refer the reader to that document. The brief description presented here is intended to give the reader enough understanding of the ANDVT LPC and the Rate Conversion algorithms to facilitate understanding this report.

An ANDVT processes voice in the manner illustrated by figure 4. On the transmit side, analog speech is A/D converted and analyzed. The analysis generates a set of ten reflection coefficients that characterize the spectral content of the voice, values for amplitude and pitch, and a binary voicing parameter. One set of these parameters is encoded into a 54 bit ANDVT frame every 22.5 ms to yield an output rate of 2400 bps. On the receive side the process is reversed and the 2400 bps input stream is used to regenerate analog voice.

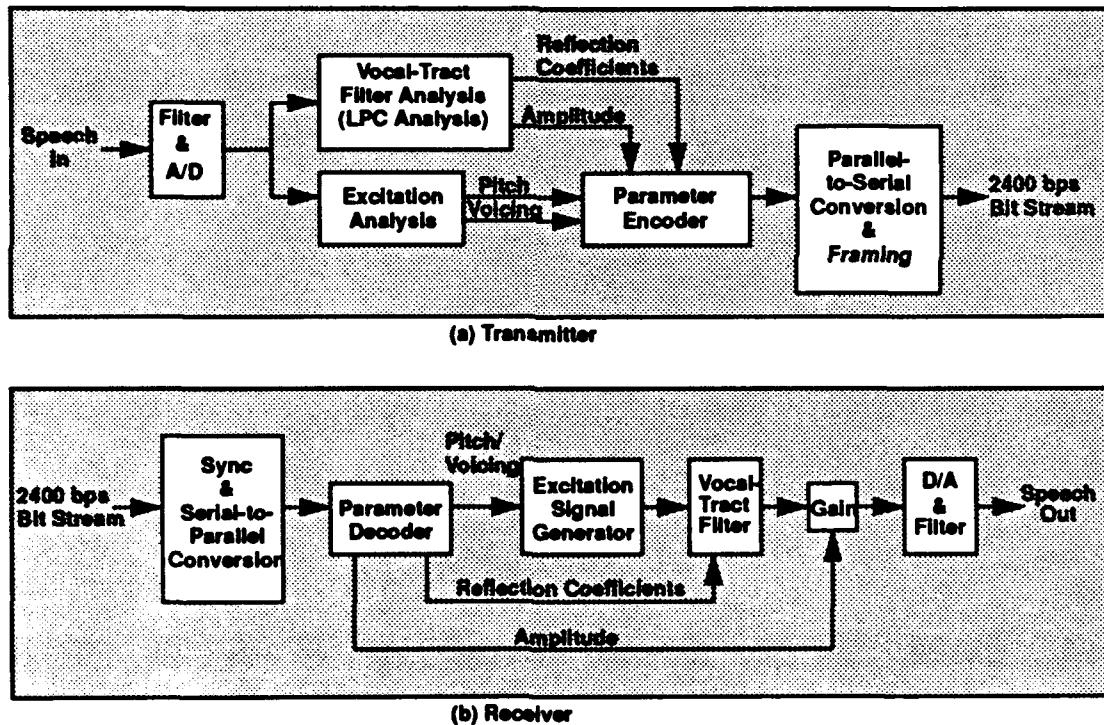


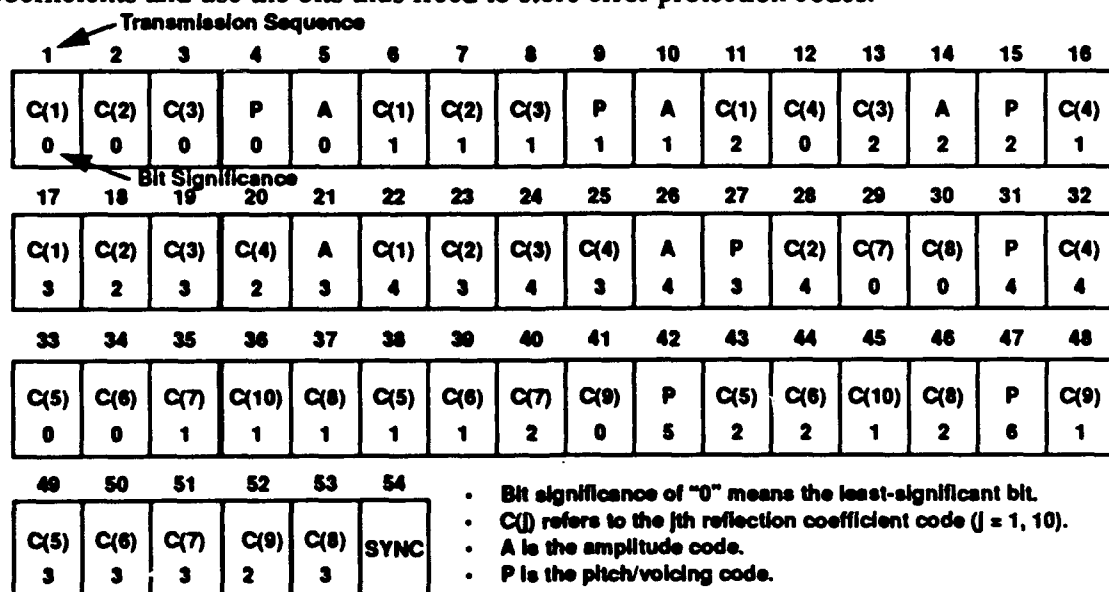
FIGURE 4. ANDVT voice processing.

The following table describes the LPC-10 parameters in more detail and indicates the number of bits allocated by the ANDVT Encoder to represent each parameter.

**Table 9: ANDVT Parameters**

Speech Parameter	Bits/Frame	Remarks
Amplitude	5	RMS value of preemphasized speech waveform quantized semi-logarithmically over a 60 dB dynamic range
Pitch	6	Quantized logarithmically from a pitch interval of 20 to 160 samples at 20 steps per octave
Voicing	1	Binary voicing decision
Reflection Coefficients	41 (Voiced frames)	The first through tenth reflection coefficients are quantized to 5, 5, 5, 5, 4, 4, 4, 4, 3, and 2 bits, respectively.
	20 (Unvoiced frames)	The first through fourth reflection coefficients are quantized to 5, 5, 5, and 5 bits, respectively. Twenty bits are used for protecting the four most significant bits of the reflection coefficients and amplitude parameters by Hamming (8,4) codes. One bit is unused.
Sync	1	Alternating "1" and "0"

ANDVT parallel-to-serial conversion and framing generates a 2400 bps bit stream comprised of contiguous frames having the formats shown in figures 5 and 6. Two formats are defined, one for voiced frames and another for unvoiced frames. Voiced frames encode a full set of ten reflection coefficients. Unvoiced frames encode only the first four reflection coefficients and use the bits thus freed to store error protection codes.



**FIGURE 5. Transmission sequence for a voiced frame.**

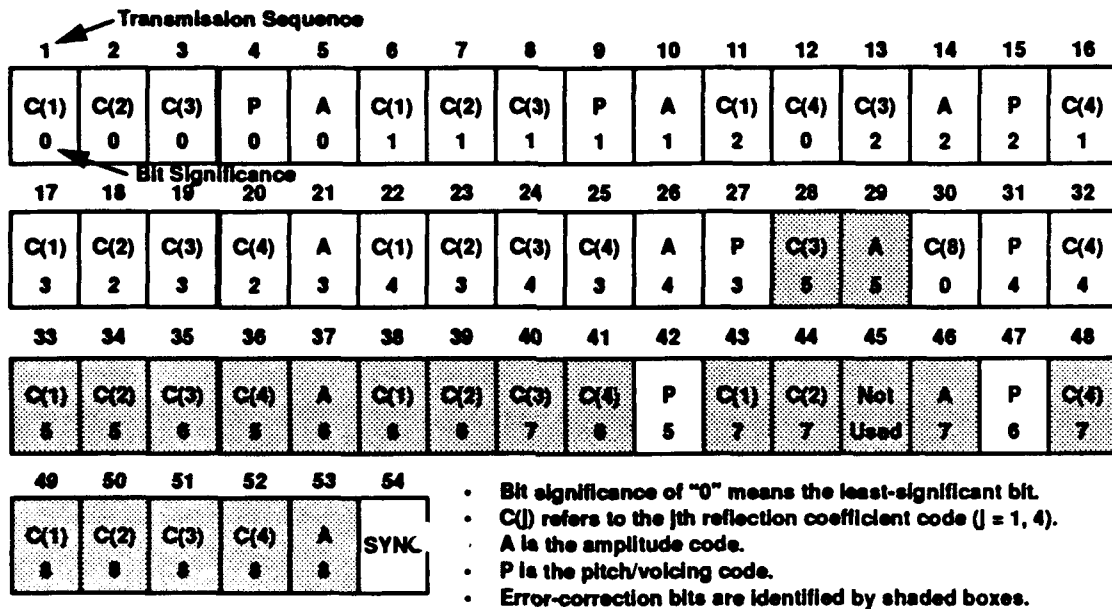


FIGURE 6. Transmission sequence for an unvoiced frame.

The rate conversion algorithm reduces four 54 bit ANDVT frames to one rate converted frame that is 108 bits in length. Elimination of one of the four ANDVT frames is the first step in rate compression. The most redundant frame is determined by comparing each of the first three frames with adjacent frames. The fourth frame must be kept because it will be adjacent to the first frame of the following set of four. This procedure reduces inter-frame redundancy.

Intra-frame redundancy is reduced by eliminating just-noticeable differences and non-speech sounds from the ANDVT's repertory of reproducible sounds. The rate conversion algorithm accomplishes this by pattern matching the large set of ANDVT patterns with a much smaller set of patterns peculiar to the human voice and only transmitting a key to represent the pattern. This vector quantization process used during compression is by far the most computationally intensive part of the rate conversion algorithm.

Rate compression and decompression are summarized in figures 7 and 8.

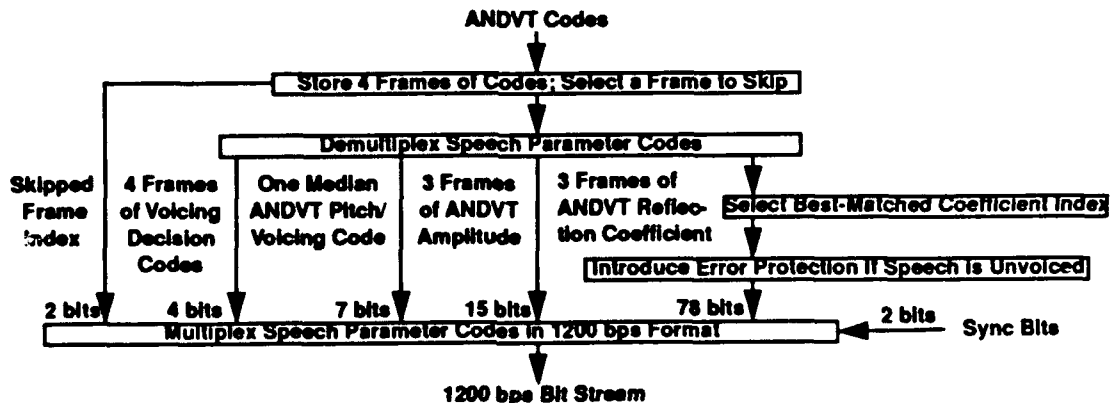


FIGURE 7. Rate conversion from 2400 bps to 1200 bps.

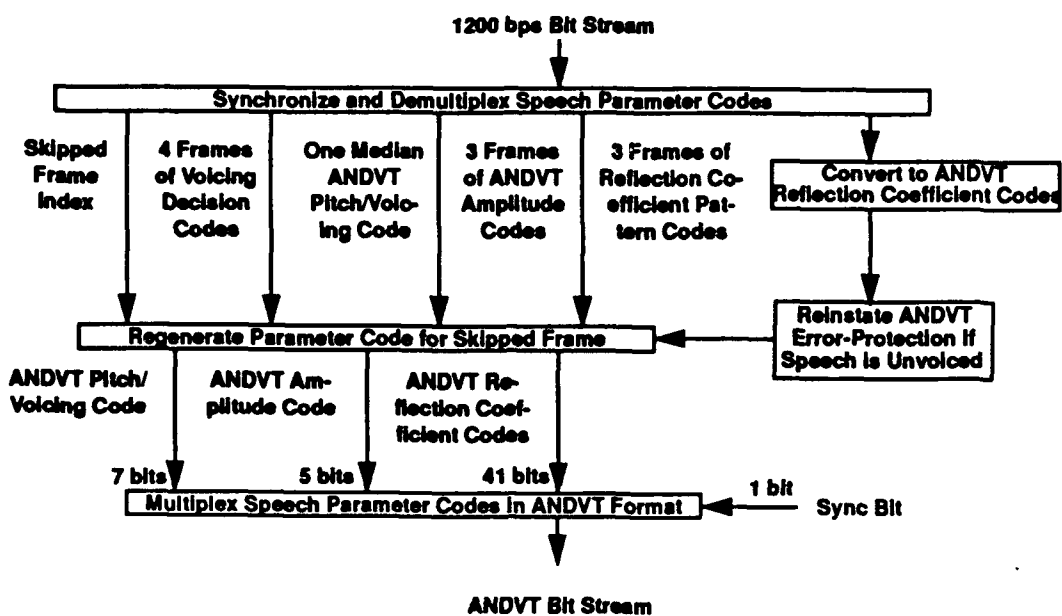
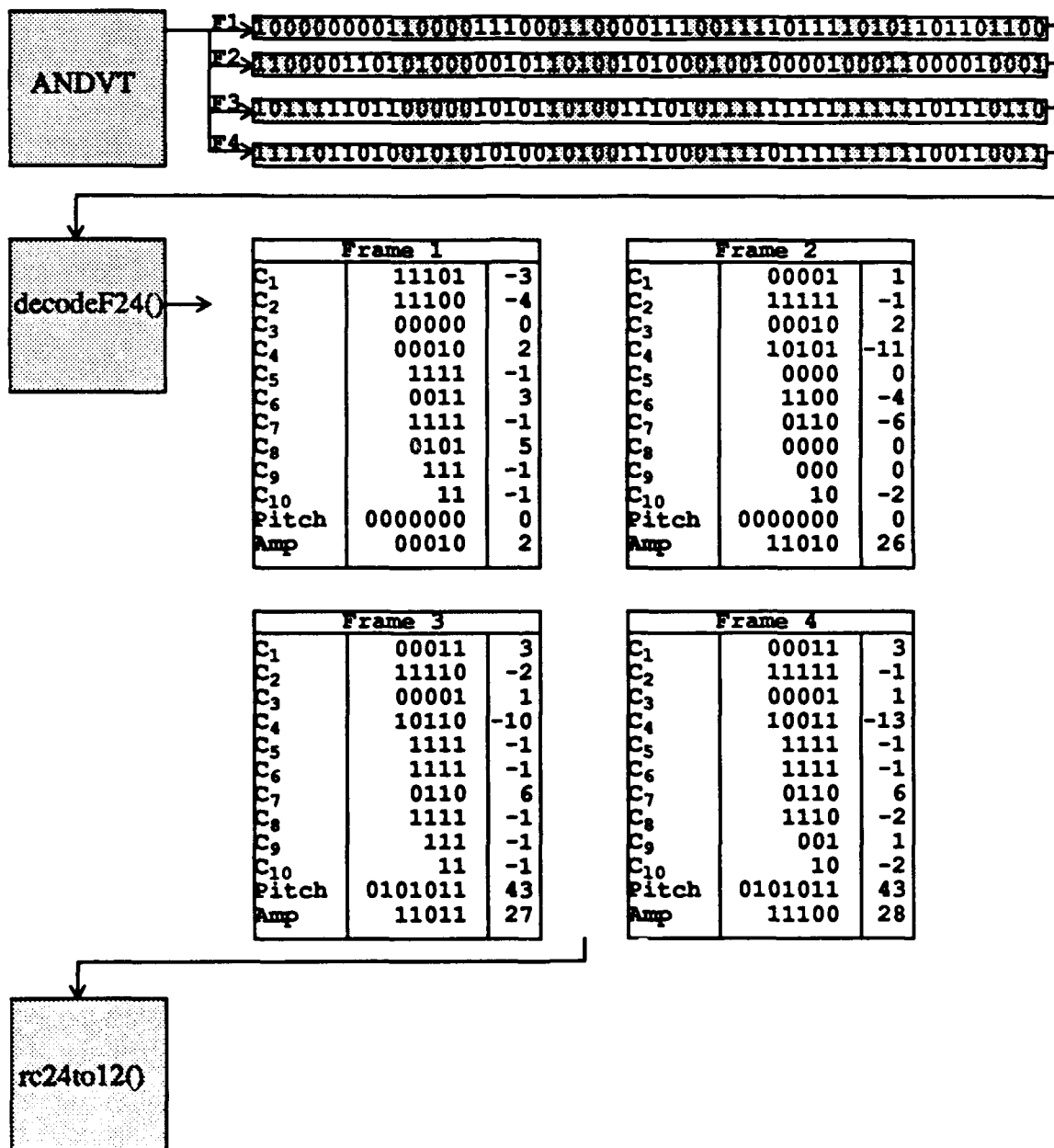


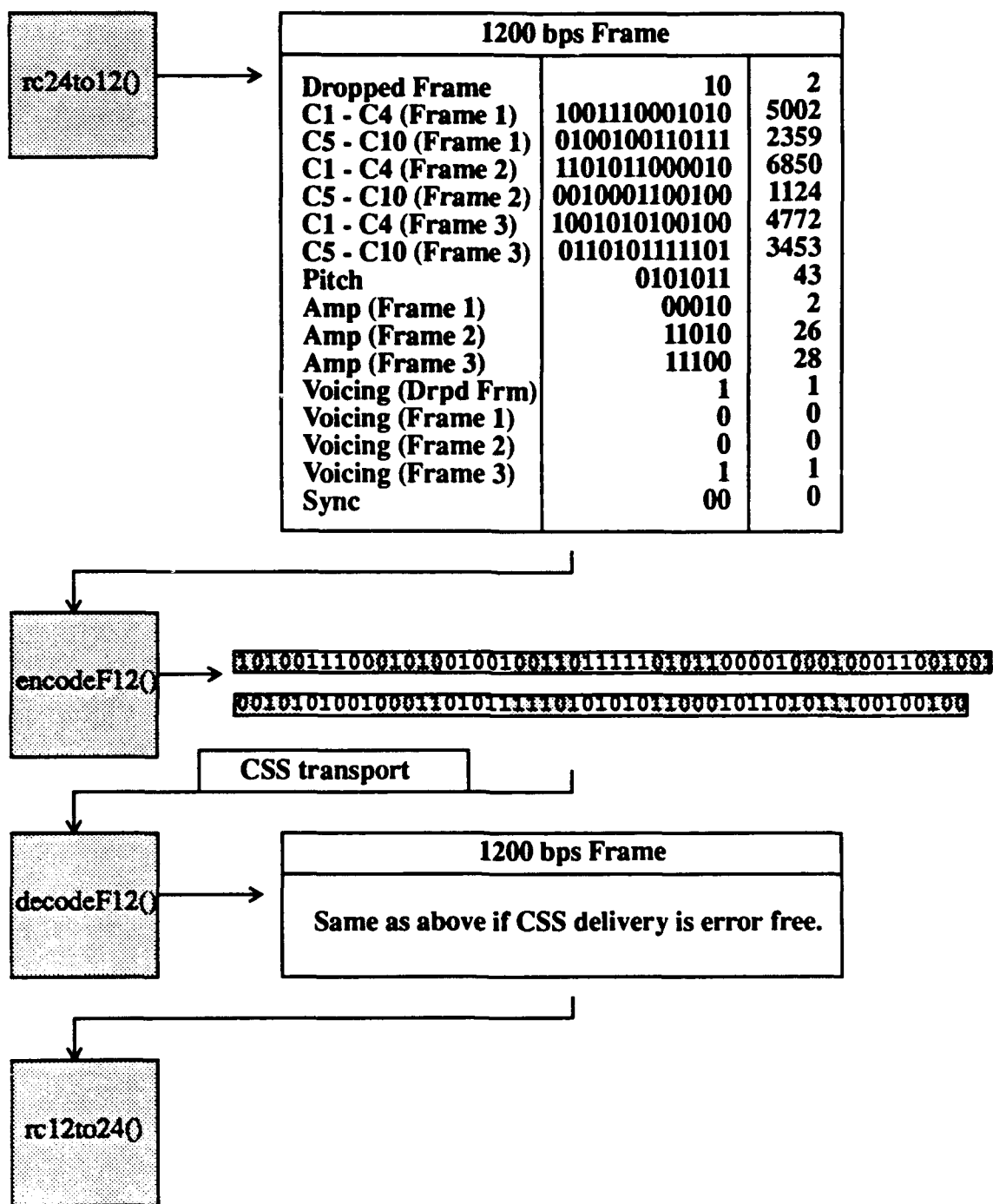
FIGURE 8. Rate conversion from 1200 bps to 2400 bps.

## 7.2 Example Compression/Decompression of Four Consecutive ANDVT Frames

In figures 9 through 11 we present the flow of data frames from an ANDVT through the rate compression and, then, the rate decompression side of the RC and back to an ANDVT.



**FIGURE 9.** An ANDVT generates four contiguous frames of 2400 bps voice data. From these frames the decodeF24() function decodes four sets of LPC-10 parameters that, in turn, are used as input for a call to rc24to12().



**FIGURE 10.** The function `rc24to12()` compresses four sets of LPC-10 parameters into one set of rate converted parameters for a 1200 bps frame. The 108 bits for this frame are encoded by a call to `encodeF12()`. After delivery to a remote RC by the CSS, the `decodeF12()` function decodes the rate converted parameters and uses them as input to a call to `rc12to24()`.

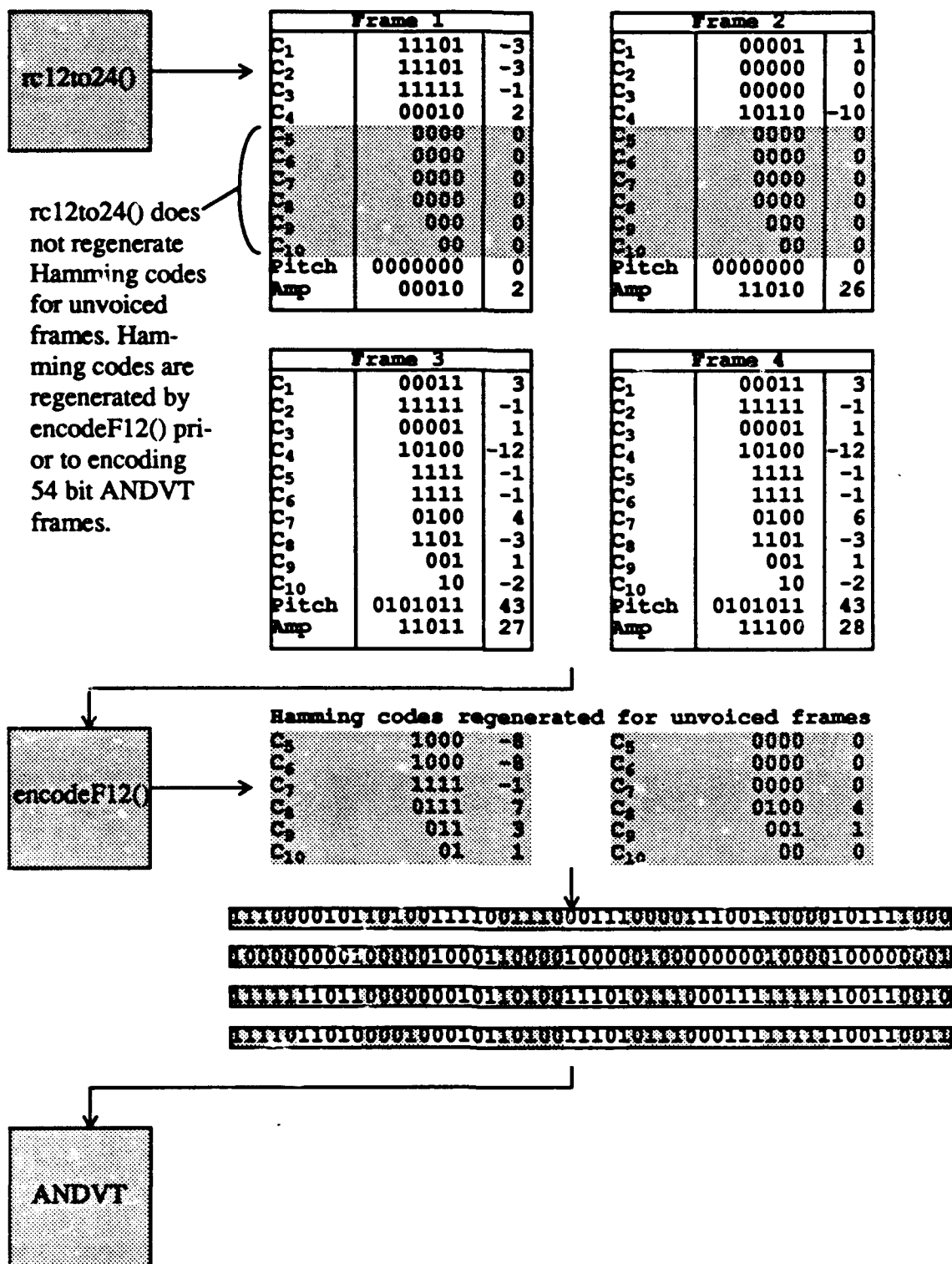


FIGURE 11. The rc12to24() function regenerates four sets of LPC-10 parameters (except for Hamming codes in unvoiced frames) from one set of rate converted input parameters. EncodeF12() is called four times, once for each set of LPC-10 parameters, to encode the parameters into four ANDVT frames. If a frame is unvoiced, Hamming codes are regenerated. Frames are output, via a VST, to an ANDVT.